UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

HERRAMIENTA GENERADORA DE TRIANGULACIONES Y VISUALIZADOR WEB

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

BENJAMÍN LUIS MELLADO CANIHUANTE

PROFESORA GUÍA:
MARÍA CECILIA RIVARA ZÚÑIGA

PROFESORES DE COMISIÓN:
FEDERICO OLMEDO BERÓN
JOSÉ MIGUEL PIQUER GARDNER

SANTIAGO DE CHILE
2023

Resumen

Este estudio se centra en la implementación práctica de una variante del algoritmo Lepp Delaunay de Rivara, denominado Lepp Casi Delaunay. Este algoritmo, utilizado para el refinamiento de triangulaciones en dos dimensiones, ha sido investigado anteriormente desde una perspectiva teórica. En este trabajo, se realizó el proceso de llevar este concepto teórico a la práctica, a través de una metodología de ingeniería de software.

Se desarrolla un software en C++ para implementar esta variante, permitiendo una generación y refinamiento de mallas de forma robusta. El software resultante incorpora operaciones de aritmética exacta y adaptativa para hacer cálculo en operaciones geométricas y garantizar la calidad de la malla. Este, se basó en el trabajo previo de Javier Díaz, quien implementó un software de refinamiento Lepp en Python.

Por otro lado, se desarrolla un visualizador web utilizando el framework web React, que ofrece una interfaz intuitiva para la interacción con las mallas generadas por el software de C++. El visualizador permite la visualización interactiva de las mallas refinadas usando una librería gráfica llamada Three.js.

Finalmente, se realizó un análisis detallado de los datos producidos por el software e imágenes del visualizador web. Este análisis permitió una evaluación en el desempeño cualitativo del algoritmo y de su aplicabilidad en el refinamiento de mallas. El trabajo del software matemático y el visualizador web desarrollados buscan contribuir en el campo de la generación y refinamiento de mallas, y ofrecer un punto de partida para posibles trabajos futuros.

Dedico este trabajo a mis padres, hermanas, profesora y a todos quienes me compartieron su cariño durante este camino.

Tabla de contenido

1. Introducción

La generación y refinamiento de triangulaciones son procesos esenciales en ingeniería y las ciencias de la computación, especialmente en áreas como la computación gráfica, la simulación numérica y la modelación geométrica. Estas triangulaciones son representaciones discretas de superficies y volúmenes, que permiten realizar cálculos y análisis computacionales en un dominio de interés. El desarrollo de nuevas técnicas de refinamiento de mallas, es tema de estudio que tiene como objetivo mejorar la eficiencia de las aplicaciones que las utilizan.

Los algoritmos *Longest Edge Path Propagation* (Lepp) desarrollados por Rivara [15, 14] pertenecen a una clase de técnicas utilizadas para la generación y refinamiento de mallas, que son esenciales en la aplicación de elementos finitos; que es una técnica numérica utilizada para la resolución de ecuaciones diferenciales parciales, las cuales surgen en una variedad de campos científicos y de ingeniería.

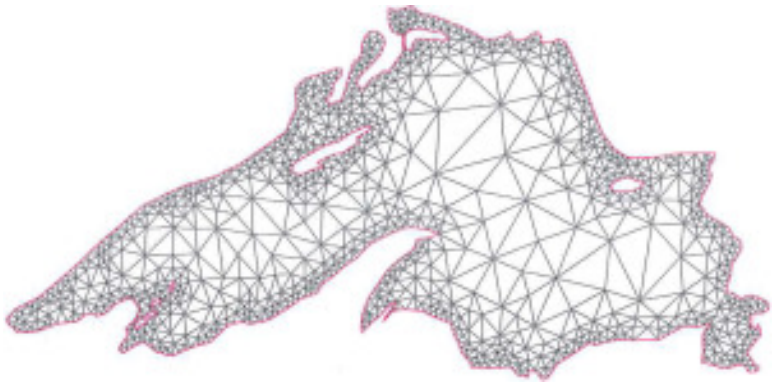


Figura 1: Geometría de un lago refinada usando el algoritmo Lepp Delaunay de Rivara.

En el año 2018, el Ingeniero Civil en Computación, Javier Díaz, desarrolló como parte de su trabajo de título [10] un software que implementó los algoritmos Delaunay de Rivara en 2D con el uso de aristas restringidas [16], todo esto utilizando el lenguaje de programación Python.

Por otra parte, el autor del presente trabajo (Mellado) y Rivara, han trabajado en una variante teórica del algoritmo Lepp Rivara con inserción Delaunay llamada Lepp Casi Delaunay [18]. Esta variante introduce una modificación en el proceso de refinamiento de la malla, de tal manera que no es necesaria la actualización recursiva de la misma cada vez que se inserta un nuevo punto para mantener la propiedad Delaunay.

De esta forma, se plantea la implementación del algoritmo Lepp Casi Delaunay. El objetivo es realizar una implementación que permita la experimentación y análisis, pudiendo contribuir al continuo desarrollo de herramientas para la generación y refinamiento de mallas.

Es importante considerar que la implementación de los algoritmos de generación y refinamiento de mallas requiere de un buen grado de precisión en los cálculos numéricos. En este contexto, la calidad de la malla resultante está relacionada con la precisión de cada operación matemática involucrada [21].

El software creado por Javier Díaz, que implementa los algoritmos Lepp de Rivara, adopta un enfoque heurístico para garantizar la robustez. Este enfoque se centra en un umbral de error para determinar el resultado de una operación aritmética [10].

En el presente trabajo, se adopta un enfoque diferente para abordar la robustez, basado en el estudio de J. Shewchuk [23]. Este enfoque intenta optimizar el equilibrio entre la precisión del cálculo y la robustez del software, adaptando la precisión de los cálculos a las demandas específicas de cada tarea.

La implementación de los algoritmos se llevó a cabo en C++, un lenguaje que ofrece un gran rendimiento y eficiencia. Además, C++ proporciona un control preciso sobre la memoria y ofrece una gran cantidad de bibliotecas matemáticas, lo que lo convierte en una elección adecuada para un software que busca ser matemáticamente robusto y preciso.

En términos de implementación de la interfaz gráfica, se ha optado por diseñar una aplicación web para la visualización y manipulación interactiva de las mallas. Esto responde a la necesidad de proporcionar un acceso visual e interactivo de las mallas. Para lograr esto, se han empleado tecnologías modernas de desarrollo web.

React [13] ha sido la elección como framework web debido a su capacidad de creación de interfaces de usuario dinámicas y extendibles, mientras que Three.js [26] se ha seleccionado como librería gráfica por su práctica funcionalidad para la renderización de gráficos 2D y 3D en el navegador.

Con este enfoque, no solo se busca proporcionar una implementación robusta y eficiente del software, si no también una plataforma interactiva de visualización para los usuarios que necesiten realizar tareas de refinamiento de mallas, desarrollando de esta forma dos herramientas distintas que pueden ser usadas

en conjunto.

Para abordar este trabajo, se proporcionará una revisión de la literatura relacionada con la generación y refinamiento de mallas, centrándose en los algoritmos de Rivara y sus modificaciones. Luego, se describirá la implementación de los algoritmos de refinamiento Lepp Delaunay y Casi Delaunay en un software de C++, incluyendo las estructuras de datos utilizadas y las bibliotecas relevantes empleadas. Finalmente, se abordará el desarrollo de la aplicación web de visualización, que permitirá a los usuarios interactuar y manipular las triangulaciones.

2. Objetivos

2.1. Descripción de los objetivos

El objetivo principal de este trabajo es la implementación de una variante del algoritmo de Lepp Delaunay de Rivara para la mejora de triangulaciones, utilizando una plataforma de software en C++ capaz de manejar cálculos matemáticos con alta precisión y estabilidad numérica. Dicha variante busca simplificar los procesos involucrados en el refinamiento de una triangulación, permitiendo su uso en una amplia variedad de aplicaciones.

Además, se revisa los trabajos previos que han hecho uso del algoritmo Lepp, para entender sus características y arquitectura. Este análisis proporciona una base para desarrollar una nueva variante del algoritmo, con optimizaciones específicas para sus necesidades.

El objetivo de implementar el algoritmo Lepp Casi Delaunay en C++ radica en explorar y abordar limitaciones identificadas en la implementación anterior realizada en Python. Aunque Python es conocido por su simplicidad y es útil en las etapas iniciales o prototípicas de un proyecto [3], C++ resalta por su eficiencia y desempeño al manejar tareas computacionalmente intensivas [25]. La elección de C++ no solo responde a su capacidad inherente de optimización, sino también a la posibilidad de integrar y aplicar herramientas específicas para una posible paralelización del algoritmo en GPU.

En el contexto académico del autor de este trabajo, se adquirió formación específica en computación en GPU, donde el lenguaje principal de instrucción fue C++. Esta decisión técnica, por lo tanto, no solo busca mejorar el desempeño del algoritmo, sino que también sienta una base robusta para futuras extensiones y optimizaciones que aprovechen capacidades específicas de paralelismo en GPU.

Por otra parte, un aspecto relevante es asegurar la validez y coherencia de los resultados obtenidos a través del software que implementa los algoritmos, por lo que se plantea crear una herramienta de visualización gráfica. Esta herramienta permitirá examinar los resultados de la triangulación de una manera más intuitiva y visual, facilitando la identificación de problemas en los resultados.

Para el desarrollo de la herramienta de visualización, se decide trabajar con bibliotecas web, dada su versatilidad y la rapidez con la que permiten prototipar aplicaciones. Además, estas bibliotecas ofrecen una gran cantidad de opciones de personalización y facilitan su integración en diversas plataformas y dispositivos.

2.2. Metodología y resultado esperado

La lógica de las clases, la arquitectura general del software y la estructura de datos empleada en el trabajo de Díaz serán tomadas en cuenta durante el desarrollo del software en C++. Aunque el software no será utilizado directamente en el presente trabajo, su enfoque y arquitectura servirán como base y guía para el diseño e implementación del software en C++.

El objetivo de esta aproximación es aplicar el conocimiento y las soluciones implementadas en el trabajo previo para construir un software en C++ que sea funcional. Se espera que esta estrategia permita una transición más fluida en el proceso de desarrollo y facilite la implementación de las modificaciones. Con el fin de adaptar el algoritmo Lepp Delaunay de Rivara al nuevo software en C++, así como la implementación del algoritmo Casi Delaunay.

Se utiliza una metodología de investigación y desarrollo que combina la teoría de los algoritmos de refinamiento de mallas, con la práctica de la implementación de software y el diseño de interfaces de usuario.

Para esto, se estudian los algoritmos de refinamiento analizando sus propiedades y características para posteriormente implementarlos en el software de C++, utilizando estructuras de datos adecuadas y bibliotecas especializadas en geometría computacional.

Finalmente, se desarrolla una aplicación web que permite visualizar e interactuar con las mallas generadas, utilizando tecnologías como React para la estructuración del código web y Three.js para la renderización de gráficos 3D en el navegador.

Se espera que este trabajo contribuya a la comprensión de los algoritmos de refinamiento de mallas y a la práctica de su implementación en el ámbito de la ingeniería en computación. Además, se espera que la aplicación web desarrollada sea útil para profesionales e investigadores que trabajan en el área de la computación gráfica, la simulación numérica y la modelación geométrica, ofreciendo una herramienta interactiva y fácil de usar para visualizar y manipular mallas refinadas.

3. Antecedentes teóricos

El refinamiento de triangulaciones es un proceso esencial en muchas aplicaciones de ingeniería y ciencias de la computación. La literatura en este campo cubre una amplia gama de algoritmos y técnicas de refinamiento [20, 6, 8].

Los algoritmos de Lepp Rivara [15, 14], son un enfoque particular para el refinamiento de mallas basado en la selección de aristas a ser divididas y la inserción de nuevos puntos. Este enfoque es especialmente útil para generar mallas adaptativas, en las que se requiere un refinamiento localizado en regiones de interés, como áreas con alta variación geométrica o en las que se necesita mayor precisión en los cálculos numéricos.

Este algoritmo tiene como función refinar mallas de manera eficiente y adaptativa, permitiendo una mejor representación de las características de la geometría del problema en cuestión. Además, el algoritmo de Lepp de Rivara puede ser fácilmente extendido a mallas en 2D y 3D, lo que lo hace especialmente útil para una amplia gama de aplicaciones.

A lo largo de los años, Rivara y otros investigadores han continuado trabajando en la mejora y adaptación de los algoritmos de Lepp-Rivara para diversas aplicaciones, como se evidencia en sus publicaciones [17, 4, 14, 19, 16].

3.1. Conceptos

3.1.1. Triangulación de Delaunay

La triangulación de Delaunay es un tipo de malla que maximiza el ángulo mínimo de todos los triángulos en la malla. La idea detrás de la triangulación de Delaunay es que un buen conjunto de triángulos es aquel que evita ángulos muy agudos. El criterio utilizado para crear estas mallas es que ningún punto de la malla debe estar dentro de la circunferencia circunscrita de cualquier triángulo en la malla [5]. Ver figura 2.

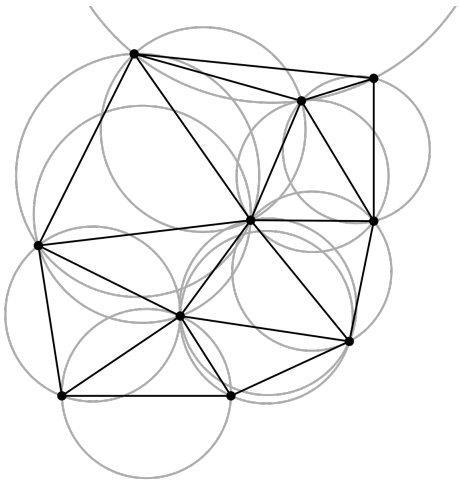


Figura 2: Triangulación de Delaunay con los circuncírculos dibujados para cada triángulo.

3.1.2. *Flip Edge*

El *Flip Edge*, también descrito como legalización de arista o intercambio de arista, es una operación dentro del contexto de una triangulación utilizada en la construcción y actualización de una triangulación de Delaunay. Al considerar dos triángulos adyacentes que comparten una arista común, el *flip* reemplaza esta arista por la otra que conecta los dos vértices opuestos, formando así dos nuevos triángulos adyacentes.

La necesidad del intercambio de arista es necesario para mantener la propiedad Delaunay en una triangulación. Según el criterio de Delaunay, ningún punto de la triangulación debe caer dentro del círculo circunscrito de cualquier triángulo en la misma triangulación. Cuando se inserta un nuevo punto o se hace alguna modificación que hace que no se cumpla esta propiedad, el intercambio de arista se emplea para restaurarla. Al intercambiar aristas, se modifica una zona local de la malla para garantizar que los triángulos adyacentes vuelvan a cumplir con el criterio de Delaunay.

En Figura 3 se ejemplifica el proceso de intercambio de arista entre los triángulos Δacb y Δabd que no cumplen la propiedad de Delaunay entre sí, al estar el punto d dentro del circuncírculo de Δacb.

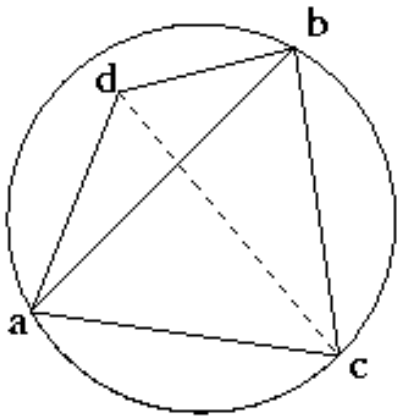


Figura 3: Intercambio de arista entre dos triángulos adyacentes Δacb y Δabd, la arista interlineada representa la nueva arista.

3.1.3. Camino de Propagación de la Arista más Larga (Lepp)

El Camino de Propagación de la Arista más Larga (*Longest Edge Propagation Path*, Lepp) es un concepto central en el algoritmo de refinamiento de malla Lepp Rivara. El Lepp, en términos generales, es una cadena de aristas en una malla que inicia en un triángulo específico y donde cada arista consecutiva en el camino es al menos tan larga como la que la precede.

Para identificar el Lepp en una malla, se inicia en un triángulo y se selecciona su arista más larga. A partir de ahí, se sigue el camino en la malla siempre seleccionando la arista más larga de los triángulos adyacentes al actual. Este proceso continúa hasta llegar a un punto donde todas las aristas adyacentes son más cortas que la arista que condujo al punto actual. La cadena de aristas que se ha seguido hasta este punto constituye el Lepp.

La última arista en la cadena del Lepp, la cual no tiene sucesora más larga o igual en longitud, se denomina arista terminal. Los triángulos que comparten esta arista terminal se conocen como triángulos terminales.

En Figura 4 se muestra el camino Lepp de un triángulo t_0, su arista terminal es AB y sus triángulos terminales son t_2 y t_3.

Los triángulos terminales son especialmente relevantes en el proceso de refinamiento de malla, ya que son los candidatos para ser divididos por la inserción de un nuevo vértice.

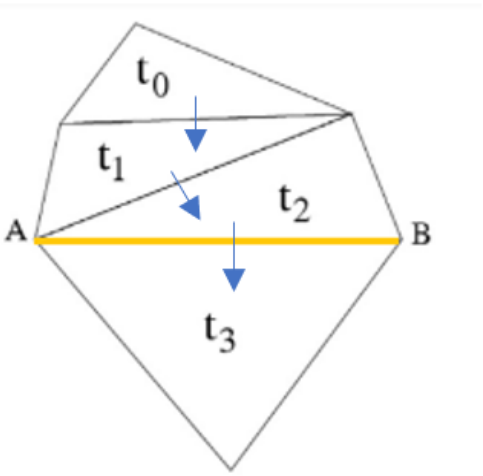


Figura 4: Representación de una arista terminal AB con triángulos terminales t_2 y t_3 a partir de un triángulo t_0.

3.1.4. Algoritmo Lepp de Rivara

El algoritmo Lepp de Rivara es una técnica de refinamiento de mallas de triangulación que se utiliza ampliamente en la geometría computacional y en la generación de mallas para simulaciones numéricas. Este algoritmo se basa en el concepto *Longest Edge Propagation Path* (Lepp) que, como se explicó anteriormente, es un camino de aristas que conecta dos vértices donde cada arista consecutiva en el camino tiene una longitud mayor o igual a la de su predecesora.

El objetivo principal del algoritmo Lepp es mejorar la calidad de las triangulaciones, enfocándose en optimizar la distribución de los ángulos en los triángulos que componen la malla. Para alcanzar este objetivo, el algoritmo inserta puntos estratégicamente en la malla. Es importante destacar que este algoritmo no se centra necesariamente en la minimización del número de triángulos, sino en mejorar su calidad de forma adaptativa.

El procedimiento se basa en un criterio de calidad definido por un ángulo de tolerancia, usualmente denotado como θ. Este valor indica que todos los triángulos de la malla deben tener todos sus ángulos mayores a esa tolerancia. Por ejemplo, si se establece una tolerancia θ de 25 grados, todos los ángulos en todos los triángulos de la malla deberán ser mayores a 25 grados.

Si durante la revisión de la malla se encuentran triángulos que no cumplen

con este criterio, llamados "triángulos malos", el algoritmo interviene para mejorarlos. Este proceso de mejora puede involucrar la división del triángulo en cuestión a lo largo de su arista más larga, creando triángulos nuevos con mejores propiedades angulares.

El algoritmo Lepp-Rivara se ejecuta a través de pasos que hacen cálculos globales y locales sobre una triangulación T (Figura 6):

1. **Identificación de triángulos malos:** Se recorre la triangulación T y se identifican todos los triángulos con ángulo mínimo menor a θ (triángulos malos).
2. **Procesamiento de triángulos malos:** Se identifica el Lepp para cada triángulo malo almacenado en la cola de forma secuencial. Para cada triángulo se ejecuta:
   - **Inserción de un nuevo vértice:** Una vez que se ha identificado el Lepp, el siguiente paso es insertar un nuevo vértice correspondiente al punto medio de la arista terminal del Lepp, que es la arista más larga. Esto da como resultado la división de los triángulos terminales en triángulos más pequeños.
   - **Actualización de la malla:** Después de la inserción del nuevo vértice, se actualiza la estructura de datos de la malla para reflejar los cambios. Esto implica actualizar las relaciones de adyacencia entre los triángulos para asegurar que la malla siga siendo una representación válida de la superficie que se está modelando.

     Para ejemplificar, en la Figura 5 se observan los pasos de actualización cuando se inserta un punto dentro de un triángulo. Estos corresponden a:
     - Eliminar $t0$.
     - Agregar $t1$, $t2$ y $t3$.
     - Actualizar adyacencias de $t1$, $t2$ y $t3$ y sus vecinos.

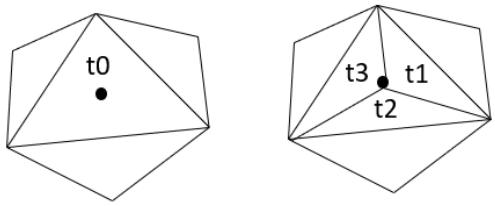


Figura 5: Actualización de una malla al insertar un punto interior.

3. **Repetición del proceso:** Finalmente, el paso 2 se repite hasta que todos los triángulos de la malla sean buenos, es decir, hasta que todos los ángulos de los triángulos sean mayores o iguales a la tolerancia, θ.

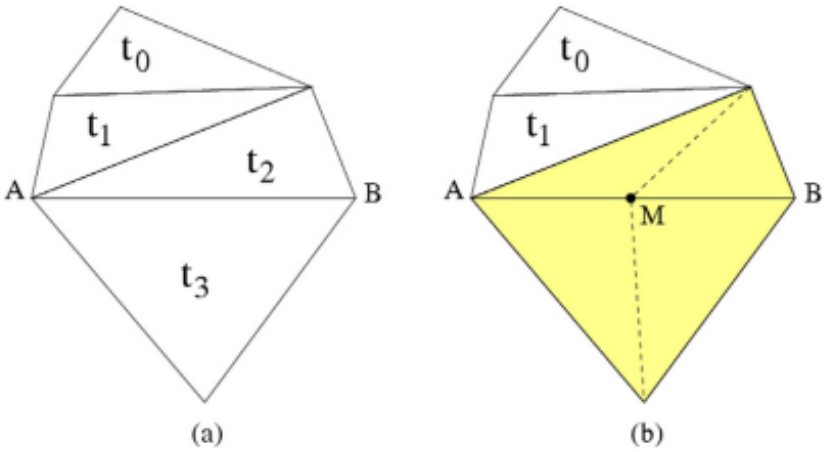


Figura 6: Algoritmo de inserción del punto medio por búsqueda del Lepp. El triángulo malo procesado es t_0.

3.1.5. Algoritmo Lepp Delaunay con inserción del centroide

El algoritmo Lepp Delaunay es una implementación del algoritmo de refinamiento de malla Lepp Rivara que mantiene la propiedad de Delaunay a lo largo de todo el proceso de refinamiento usando la inserción del centroide [14]. En cada iteración, el algoritmo Lepp Delaunay inserta un nuevo punto en la malla y luego lleva a cabo un proceso recursivo de legalización de aristas para garantizar que se mantenga la propiedad de Delaunay.

El proceso *Flip Edge* se lleva a cabo de la siguiente manera: después de cada inserción de un nuevo punto, todos los triángulos que comparten este punto se verifican para comprobar si satisfacen la condición de Delaunay. Si se encuentra un par de triángulos adyacentes que no cumple con la condición, se lleva a cabo un intercambio de arista y el proceso de legalización se aplica recursivamente a los triángulos nuevos formados por el intercambio.

Funcionamiento del algoritmo Lepp Delaunay (Figura 7):

1. **Identificación de triángulos malos:** Se realiza una revisión completa de la triangulación T para identificar todos los triángulos cuyo ángulo mínimo sea menor a θ (triángulos malos).
2. **Procesamiento de triángulos malos:** Se identifica el Lepp para ca-

da triángulo malo almacenado en la cola de forma secuencial. Para cada triángulo se ejecuta:

- **Inserción de un nuevo punto:** Una vez identificados los triángulos terminales, se procede a la inserción del centroide que se calcula como el promedio de los 4 vertices de los triángulos terminales que comparten la arista terminal. Si el triángulo está en el borde o su segunda arista más larga es un borde, se inserta el punto medio de la arista más larga. En caso contrario, se calcula y se inserta el punto de inserción (centroide) en la triangulación T.
- **Actualización de la malla:** Tras la inserción del nuevo punto, se legalizan las aristas de los nuevos triángulos que contienen el punto insertado. Esto implica actualizar las relaciones de adyacencia entre triángulos para asegurar que la malla siga siendo una representación válida de la superficie que se está modelando.

3. **Repetición del proceso:** Finalmente, se repite el paso 2 hasta que todos los triángulos sean buenos, es decir, hasta que todos los ángulos en los triángulos sean mayores o iguales a la tolerancia, θ.

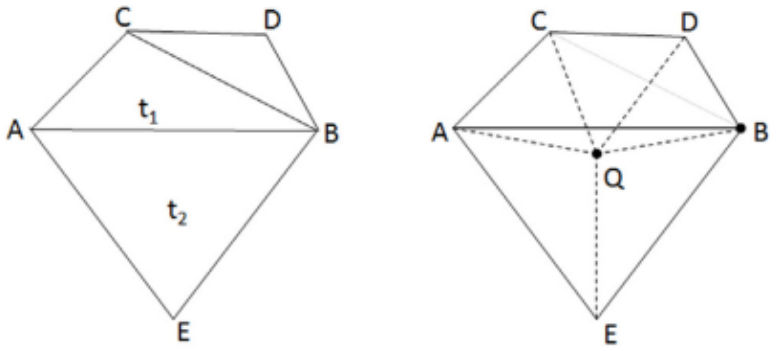


Figura 7: Algoritmo de inserción del centroide por búsqueda del Lepp, a partir de un triángulo malo ΔCBD.

Este algoritmo garantiza que la malla resultante sea una triangulación de Delaunay, lo que puede ser beneficioso en aplicaciones que requieren esta propiedad. A continuación se muestra la implementación en pseudocódigo.

Algoritmo 1: Algoritmo Lepp Delaunay con inserción del centroide

Resultado: Refinamiento de la malla hasta que todos los triángulos sean buenos

```
t_malos = Recorrer la triangulación T y encontrar todos los triángulos
 con ángulo mínimo menor a θ (triángulos malos);
for t en t_malos do
    while angulo_minimo(t) <θ do
        ta, tb = seleccionar triángulos terminales basado en la cadena
         Lepp;
        if arista terminal es borde o ta tiene su segunda arista más larga
         en el borde then
            p = calcular el punto medio de la arista más larga en t;
        else
            p = calcular el centroide de los triángulos terminales;
        end
        insertar p en T;
        for arista e en triangulos nuevos que contengan p do
            legalizarArista(T, e);
        end
    end
end
```

3.1.6. Algoritmo Lepp Casi Delaunay

El algoritmo Lepp Casi Delaunay se presenta como una variación del algoritmo de refinamiento de malla Lepp Delaunay, con una particularidad distintiva en su manejo de los triángulos malos [18]. Este algoritmo no realiza automáticamente una legalización de aristas después de cada inserción de punto. En cambio, usa dos alternativas cuando se encuentra con un triángulo malo y llega a la arista terminal de su Lepp.

La primera alternativa se ejecuta si los triángulos terminales son localmente Delaunay entre sí; en ese caso, el algoritmo procede a insertar el centroide. La segunda alternativa se despliega si los triángulos terminales no son localmente Delaunay; en esta situación, el algoritmo realiza un intercambio de aristas para que los triángulos se conviertan en Delaunay entre sí.

Es importante señalar que, en ambas alternativas, el algoritmo actualiza la malla. Esto implica que, independientemente de si se ha insertado un punto o se ha realizado una legalización de aristas, se debe calcular nuevamente el Lepp si

el triángulo malo persiste. Esta característica garantiza la optimización continua de la malla hasta que se cumplan las condiciones de calidad establecidas.

Funcionamiento del algoritmo Lepp Casi Delaunay:

1. **Identificación de triángulos malos:** Se recorre la triangulación T y se identifican todos los triángulos con ángulo mínimo menor a θ (triángulos malos).
2. **Procesamiento de triángulos malos:** Se identifica el Lepp y sus triángulos terminales para cada triángulo malo almacenado en la cola de forma secuencial. Para cada triángulo se ejecuta:
   - **(Opción 1) Intercambio de aristas:** Si los triángulos terminales no son localmente Delaunay, se realiza un intercambio de aristas para legalizar la arista compartida entre los dos triángulos terminales.
   - **(Opción 2) Inserción del punto:** Si el triángulo es un borde o tiene su segunda arista más larga en el borde, se inserta el punto medio de la arista más larga. De lo contrario, se calcula e inserta el punto de inserción (centroide o punto medio) en la triangulación T.
   - **Actualización de la malla:** Se inserta el nuevo punto haciendo una división del polígono terminal en 4 triángulos (split_into_four).
3. **Repetición del proceso:** Se repite el paso 2 hasta que todos los triángulos sean buenos.

La impplementación del algoritmo en pseudocódigo:

Algoritmo 2: Refinamiento Lepp casi Delaunay

Resultado: Refinamiento de la malla hasta que todos los triángulos sean buenos

```
t_malos = Recorrer la triangulación T y encontrar todos los triángulos
 con ángulo mínimo menor a θ (triángulos malos);
for t en t_malos do
    while angulo_minimo(t) <θ do
        ta, tb = seleccionar triángulos terminales basado en la cadena
         Lepp;
        if triángulos terminales ta y tb no son localmente Delaunay then
            intercambiar aristas;
        else
            if arista terminal es borde o ta tiene su segunda arista más
             larga en el borde then
                p = calcular el punto medio de la arista más larga en t;
            else
                p = calcular el punto de inserción (centroide o punto
                 medio);
            end
            insertar p en T;
        end
    end
end
```

3.1.7. *Planar Straight Line Graph* (PSLG) y su representación en archivo de texto

Un *Planar Straight Line Graph* (PSLG) es una representación geométrica de un conjunto de puntos y líneas en el plano. En un PSLG, los puntos son vértices y las líneas son segmentos de línea recta entre pares de vértices. La triangulación de un PSLG da como resultado un conjunto de triángulos, donde cada triángulo consta de tres vértices que son conectados por tres aristas.

La representación de una triangulación PSLG en un archivo de texto se organiza en tres secciones principales: los vértices, los triángulos y sus triángulos vecinos. Cada sección se indica con una etiqueta inicial ("v" para vértices, "t" para triángulos y "n" para vecinos).

La sección de vértices enumera todas las coordenadas de los vértices en el formato "v x y", donde "x" e "y" son las coordenadas del vértice.

La sección de triángulos enumera todos los triángulos formados por los vértices en el formato “t v1 v2 v3”, donde “v1”, “v2” y “v3” son los índices de los vértices que forman el triángulo.

La sección de vecinos enumera todos los vecinos de cada vértice en el formato “n v1 v2 ... vn”, donde “v1”, “v2”, ..., “vn” son los índices de los vértices que son vecinos del vértice correspondiente.

Un ejemplo de representación de una triangulación PSLG en un archivo de texto se muestra a continuación:

```
Vertices
v x1 y1
v x2 y2
...
v xn yn

Triangles
t v1 v2 v3
...
t vn-2 vn-1 vn

Neighbors
n v1 v2 ... vn
...
n vn-1 vn v1
```

En la entrada, el software matemático utiliza este archivo para leer un conjunto de vértices, la triangulación que forman estos vértices y los vecinos de cada triángulo. Una vez que los datos se cargan en el software, se pueden utilizar para varias operaciones de cálculo y visualización, como la representación gráfica del PSLG, la realización de análisis de mallas, entre otros.

En la salida, el software matemático puede generar este tipo de archivos para representar los resultados de sus cálculos. Por ejemplo, después de realizar ciertas transformaciones en un PSLG, el software puede escribir el PSLG transformado en este formato de archivo para su posterior análisis o visualización.

3.1.8. Aristas Restringidas

Las aristas restringidas se refieren a aquellos segmentos de línea que deben ser respetados durante el proceso de refinamiento o generación de mallas. Estas aristas suelen representar características importantes o límites en una geometría, como contornos o divisiones entre distintos materiales. Una triangulación de Delaunay con aristas restringidas se le conoce como Triangulación de Delaunay Restringida.

3.1.9. Operaciones geométricas en mallas

Las operaciones geométricas son un componente estructural en el procesamiento geométrico, y específicamente, en los algoritmos de refinamiento de mallas. Las dos operaciones más fundamentales en este contexto son *Orient2D* e *InCircle2D*.

Orient2D

La operación *Orient2D* determina la orientación relativa de tres puntos en el plano, permitiendo determinar si un punto se encuentra sobre una recta, a la izquierda o derecha. Esta operación es esencial para la creación, búsqueda y manipulación de triángulos en una malla.

InCircle2D

La operación *InCircle2D* determina si un punto se encuentra dentro del círculo circunscrito de un triángulo. Esta operación es usada en los algoritmos de construcción y refinamiento de mallas, como el algoritmo Lepp Delaunay. Con el resultado de esta operación se determina si dos triángulos son Delaunay o es necesario hacer un intercambio de aristas.

3.1.10. Aritmética exacta y adaptativa

La aritmética exacta y adaptativa se utiliza en geometría computacional para evitar los problemas que pueden surgir debido a errores de redondeo y precisión limitada en la aritmética de punto flotante estándar [21, 23]. Cuando se realizan cálculos geométricos, en ocasiones es relevante que estos cálculos sean exactos y robustos, ya que pequeños errores pueden tener efectos en el resultado final. Por ejemplo, al calcular la intersección de dos líneas o al determinar si un punto se encuentra dentro de un polígono, un error de redondeo podría llevar a resultados incorrectos.

La aritmética exacta garantiza que los resultados de los cálculos sean precisos hasta el último bit, eliminando los errores de redondeo. Sin embargo, la aritmética exacta puede ser computacionalmente costosa.

Por otra parte, la aritmética de precisión adaptativa proporciona un equilibrio entre la precisión y el rendimiento, al adaptar la precisión del cálculo a la tarea en cuestión. Esto permite realizar cálculos exactos cuando es necesario, sin incurrir en el costo computacional de la aritmética exacta para todos los cálculos. Aún así, la aritmética adaptativa puede fallar en algunos pocos casos.

3.2. Estado del arte

En esta sección se hablará sobre tres alternativas de software de refinamiento en 2D actual. De los cuales, los dos primeros cuentan con funcionalidades y arquitectura que servirán como base para construir el software de este trabajo.

3.2.1. Software de Javier Díaz

En el ámbito del software para implementar los algoritmos de Rivara en 2D se encuentra la implementación de Javier Díaz en Python [10]. Su software implementa los algoritmos Lepp Delaunay a partir de una PSLG [16].

Además de su capacidad para manejar los algoritmos de Rivara, el software de Díaz también proporciona soporte para el uso de aristas restringidas en un PSLG. Esta característica es especialmente útil cuando se necesita preservar ciertas características de la geometría del problema, que no deben ser modificadas por el algoritmo de malla.

El software de Díaz es autocontenido, es decir que todo el código y las bibliotecas necesarias para su funcionamiento están incluidos en el propio software. Además, incluye un visualizador incorporado basado en la biblioteca Tkinter de Python, que permite visualizar la malla resultante de una manera intuitiva y sencilla.

En cuanto a robustez, se utiliza un umbral de error para determinar el resultado de las operaciones en la malla como *Orient2D* o *InCircle2D* sin uso de aritmética exacta.

Como limitación, para el manejo de la estructura, el software utiliza listas de vértices para almacenar la malla sin un manejo directo de la memoria. Esto puede llevar a duplicacion innecesaria de listas, lo cual puede afectar el rendimiento

del software.

3.2.2. Software de Shewchuck: *Triangle*

Triangle es un software ampliamente utilizado en el campo de la geometría computacional desarrollado por Shewchuk en C [24]. Su principal función es la generación y refinamiento de triangulaciones en 2D.

Triangle implementa como base los algoritmos de generación y refinamiento de mallas de Ruppert [20] y de Chew [8].

Un aspecto del diseño de Triangle es el uso de aritmética de precisión adaptativa para realizar las operaciones *Orient2D* e *InCircle2D* [23]. Que son las mismas operaciones de cálculo que se usarán para este trabajo.

3.2.3. Librería CGAL

CGAL es una biblioteca de código abierto que ofrece una serie de algoritmos para problemas de geometría computacional en geometrías 2D y 3D [7]. Esta librería proporciona herramientas como la determinación de la orientación de los puntos en un plano y generación y refinamiento de triangulaciones de Delaunay.

La implementación del algoritmo de refinamiento 2D de CGAL se basa en el trabajo de Shewchuck que a su vez extiende una implementación de Ruppert [22].

4. Arquitectura e implementación del software de C++

4.1. Estructuras de datos y clases

4.1.1. Estructura de datos basada en caras de triángulos

La representación de la malla en el software desarrollado se basa en una estructura de datos centrada en los triángulos, es decir, se utiliza una clase `Triangle` que contiene información sobre sus vértices. Esta estructura de datos permite una representación más sencilla y eficiente de las relaciones entre los triángulos y sus vértices.

En esta estructura los triángulos se representan como un conjunto de 3 vértices que se almacenan en sentido antihorario o *counterclockwise*. También se almacenan los vecinos de cada triángulo (a lo más 3) respetando el orden antihorario y con su índice equivalente al vértice opuesto del vecino. Por ejemplo, en Figura 8 la referencia del triángulo vecino opuesto al vértice i se almacena en la posición i.

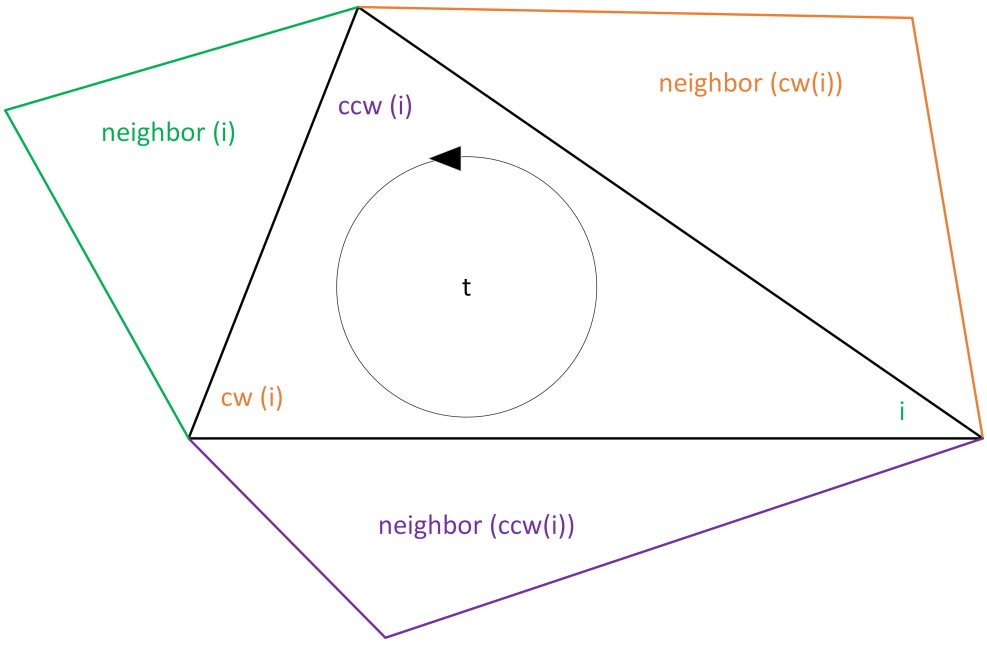


Figura 8: Representación de la estructura de datos basada en los triángulos. Los términos *cw* y *ccw* se refieren a horario y antihorario respectivamente.

4.1.2. Clases y lógica

En esta subsección, se describen las principales clases utilizadas en el software previo de Díaz y la lógica detrás de ellas:

Point: Esta clase representa un punto en el espacio bidimensional. Almacena sus coordenadas x e y y proporciona métodos para realizar operaciones geométricas básicas, como calcular distancias y ángulos entre puntos.

Triangle: La clase Triangle representa un triángulo en la malla y contiene tres objetos Point, que son sus vértices. Además, mantiene referencias a sus tres triángulos vecinos, lo que permite navegar por la estructura de la malla de manera efectiva.

Triangulation: Esta clase es la responsable de mantener y gestionar la estructura completa de la malla. Contiene una colección de objetos Triangle que conforman la malla y proporciona métodos para agregar y eliminar triángulos, así como para realizar búsquedas y navegación en la estructura. También implementa los algoritmos de refinamiento y las operaciones necesarias para mantener la calidad de la malla a medida que se modifican los triángulos y sus relaciones.

Estas clases forman la base de la implementación del software y permiten representar y manipular la estructura de la malla.

4.2. Adaptación del Algoritmo de Refinamiento Lepp Delaunay

El proceso de adaptar las funcionalidades de Python a C++ implicó una serie de consideraciones técnicas, que se describen a continuación:

1. **Tipos de Datos:** Python es un lenguaje de tipado dinámico lo que implica que no es necesario especificar el tipo de una variable cuando se declara. En contraste, C++ es un lenguaje de tipado estático, lo que requiere que se especifique el tipo de cada variable antes de su uso. Durante la adaptación, se revisó el código y asignaron tipos apropiados a todas las variables.
2. **Estructuras de Datos:** Las estructuras de datos en C++ son más concretas y menos flexibles que en Python. Por ejemplo, las listas y los diccionarios en Python pueden almacenar cualquier tipo de datos. En cambio, las estructuras de datos equivalentes en C++, como los vectores y los mapas, son más rígidos y solo admiten un tipo de valor.

3. **Uso de Punteros:** Python maneja automáticamente la memoria y la referencia de objetos, por lo que no es necesario preocuparse por los punteros. Por otro lado, C++ proporciona un control directo de la memoria a través del uso de punteros, lo que puede conducir a un código más eficiente, pero también más complejo. Se tuvo que utilizar punteros en nuestro código de C++ para manejar eficientemente la memoria y las referencias a los objetos.
4. **Funciones y Clases:** Ambos lenguajes soportan la programación orientada a objetos y ofrecen características similares en términos de funciones y clases. Sin embargo, se adaptaron las definiciones de funciones y clases para asegurar de que se ajustaran a la sintaxis y a las convenciones de C++.

4.3. Arquitectura de los directorios

El software en C++ utiliza CMake [9] como sistema de construcción y los encabezados de Shewchuck para las operaciones geométricas. La organización de carpetas y archivos se diseñará para facilitar la navegación, mantenimiento y expansión del proyecto.

La estructura de directorios propuesta es la siguiente:

- `src/`
  - `classes/` - Contiene las clases principales del software.
  - `utilities/` - Funciones auxiliares y utilidades generales.
- `test/` - Pruebas unitarias.
- `data/`
  - `input/` - Archivos de entrada para las mallas.
  - `output/` - Archivos de salida generados por el software.
- `include/` - Encabezados con funciones de Shewchuck
- `CMakeLists.txt` - Archivo de configuración de CMake.

A continuación se muestra una captura del entorno de desarrollo del software usando el IDE CLion [12]:

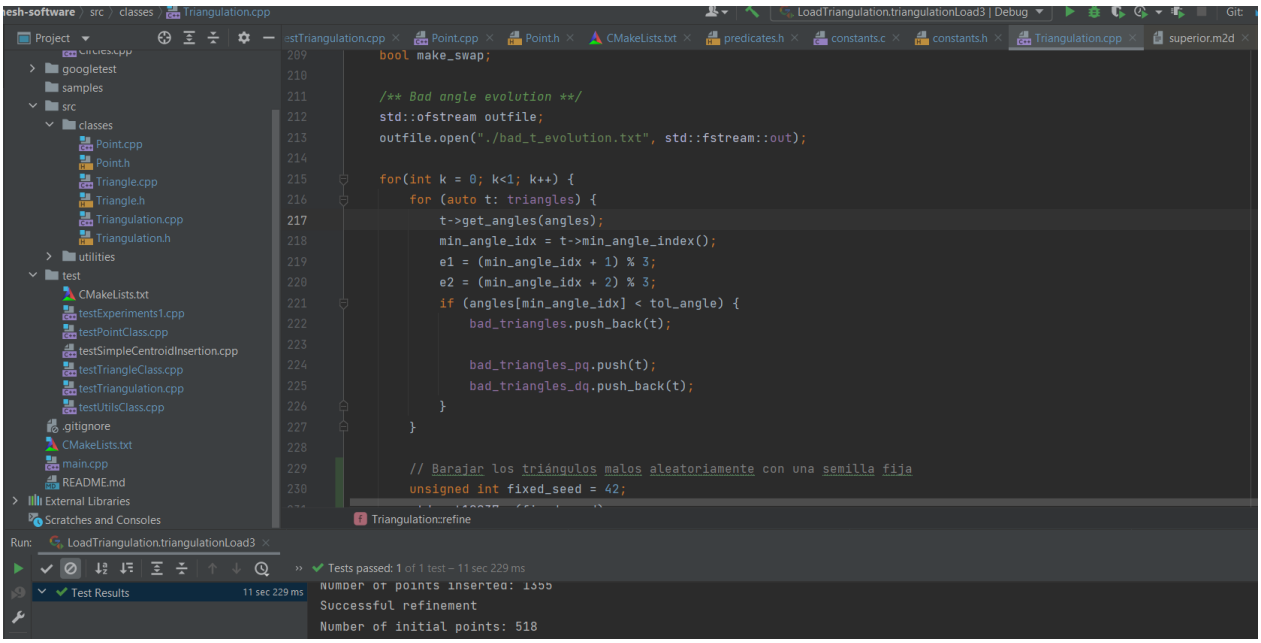


Figura 9: Captura del ambiente de desarrollo de la aplicación. Se trabajó usando el IDE CLion.

Para este software considerando las clases, archivos auxiliares y archivos de test se desarrollaron aproximadamente 2045 líneas de código.

4.4. Estructura del código

Los *headers* (.h) contienen las declaraciones de las clases, mientras que los archivos de implementación (.cpp) contienen las definiciones de las funciones miembro de las clases. El directorio principal incluye el archivo CMakeLists.txt para facilitar la compilación y la generación de ejecutables.

A continuación se describe el contenido de los *headers* con la definición de clases y métodos más relevantes en pseudocódigo. Para los métodos que no retornan un valor se describe su funcionalidad en el apartado acción.

4.4.1. Contenido de `Point.h`

Clase: `Point`

Variables:

- `m_x`: Almacena la coordenada x del punto como número de tipo `double`.
- `m_y`: Almacena la coordenada y del punto como número de tipo `double`.

Métodos:

- `Point(x, y)`: Constructor de la clase.
  - **Recibe**: Coordenadas `x` e `y` del punto.
  - **Acción**: Asigna los valores recibidos a las variables miembro `m_x` y `m_y`.
- `getCoordinates()`: Obtiene las coordenadas del punto.
  - **Retorna**: Un par de números `(x, y)` representando las coordenadas del punto.
- `setPoint(x, y)`: Actualiza las coordenadas del punto.
  - **Recibe**: Coordenadas `x` e `y`.
  - **Acción**: Actualiza las variables `m_x` y `m_y` con los valores recibidos.
- `isEqualTo(p)`: Comprueba si dos puntos son iguales.
  - **Recibe**: Un objeto `p` de tipo `Point`.
  - **Retorna**: `True` si el punto actual y el punto `p` son iguales, `False` en caso contrario.

4.4.2. Contenido de `Triangle.h`

Clase: `Triangle`

Variables:

- `points[3]`: Un array de punteros a objetos `Point` que representan los vértices del triángulo.
- `neighbors[3]`: Un array de punteros a objetos `Triangle` que representan los triángulos vecinos.

Métodos:

- `Triangle(p1, p2, p3)`: Constructor de la clase.
  - **Recibe**: Tres puntos (`p1, p2, p3`) que representan los vértices del triángulo.
- `getPoints()`: Obtiene los vértices del triángulo.
  - **Retorna**: Un puntero a un array constante de punteros a objetos `Point` que contienen los vértices del triángulo.
- `setPoints(p1, p2, p3)`: Establece los vértices del triángulo.
  - **Recibe**: Tres puntos (`p1, p2, p3`).

  - **Acción**: Actualiza los puntos del triángulo con los valores recibidos.
- `setNeighbors(t1, t2, t3)`: Establece los triángulos vecinos.
  - **Recibe**: Tres triángulos (`t1, t2, t3`).
  - **Acción**: Actualiza los vecinos del triángulo con los valores recibidos.
- `splitIntoThree(p, new_triangles)`: Divide el triángulo en tres nuevos triángulos dado un punto de división.
  - **Recibe**: Un punto `p` y un array `new_triangles` para almacenar los nuevos triangulos.
  - **Acción**: Divide el triángulo actual en tres nuevos triángulos usando el punto `p` como vértice para los nuevos triángulos.
- `splitIntoFour(p, a, new_triangles)`: Divide dos triángulos en cuatro nuevos triángulos dado un punto de división.
  - **Recibe**: Un punto `p`, un índice al triángulo vecino `a` y un array de nuevos triángulos `new_triangles` para almacenarlos.
  - **Acción**: Divide el triángulo actual y su vecino en la posición `a` en cuatro nuevos triángulos usando el punto `p` como vértice para los nuevos triángulos.

4.4.3. Contenido de `Triangulation.h`

Clase: Triangulation

Variables:

- `points`: vector de puntos en la triangulación.
- `triangles`: vector de triángulos en la triangulación.
- `edges`: lista de aristas bordes en la triangulación.
- `tol_angle`: ángulo de tolerancia utilizado para el refinamiento de la triangulación.
- `bad_triangles`: vector que almacena triángulos malos durante el proceso de refinamiento.

Métodos:

- `Triangulation()`: Constructor de la clase que inicializa la triangulación con un gran triángulo como contenedor.
  - **Acción**: Inicializa la triangulación con un gran triángulo como contenedor.
- `insertPoint(p)`: Inserta un punto en la triangulación.

  - **Recibe**: Un punto `p`.
  - **Acción**: Inserta el punto `p` en la triangulación.
- `findTriangle(p)`: Encuentra el triángulo contenedor de un punto.
  - **Recibe**: Un punto `p`.
  - **Retorna**: El triángulo que contiene al punto `p`.
- `legalizeEdge(t, a)`: Legaliza la arista de un triángulo con su vecino.
  - **Recibe**: Un triángulo `t` y un entero correspondiente al vecino `a`.
  - **Acción**: Legaliza el borde del triángulo `t` en la posición `a`.
- `lepp(t)`: Calcula el camino de borde más largo desde un triángulo dado.
  - **Recibe**: Un triángulo `t`.
  - **Retorna**: Una lista de triángulos que forman el camino de borde más largo desde `t`.
- `refine(tol_angle, simpleInsertion, centroid)`: Refina la triangulación.
  - **Recibe**: Un ángulo de tolerancia `tol_angle` y dos booleans opcionales `simpleInsertion` y `centroid`.
  - **Acción**: Refina la triangulación utilizando el ángulo de tolerancia y las opciones de inserción simple (algoritmo casi delaunay) y/o con centroide.

4.5. Bibliotecas externas: Operaciones de aritmética adaptativa de Shewchuk

Una parte importante de la implementación del software de triangulaciones en C++ es la inclusión de la biblioteca de operaciones con aritmética adaptativa de Shewchuk. Esta biblioteca de dominio público, originalmente escrita en C, ha sido adaptada a una versión en C++ que se integra fácilmente con el código existente. La biblioteca proporciona dos funciones esenciales, `orient2D()` e `incircle2D()`.

De esta librería las funciones de interés están definidas e implementadas dentro de los archivos predicates.h y predicates.cpp respectivamente.

4.6. Pruebas de unitarias de testeo

Para verificar la correcta funcionalidad de las clases implementadas en este proyecto, se desarrollaron pruebas de unidad utilizando Google Test [11]. Estas pruebas consisten en pequeños escenarios diseñados para comprobar aspectos

específicos del código, como el correcto funcionamiento de una clase o método en particular. Los casos de prueba representan situaciones normalmente encontradas durante la ejecución del programa y garantizan que el software se comporta como se espera en tales situaciones.

A continuación, se presentan algunos de los casos de prueba con los que se testeo el software:

Prueba de igualdad de puntos en un triángulo

Esta prueba valida si dos triángulos son idénticos basándose en la igualdad de sus puntos. Se crean dos triángulos con los mismos puntos pero en distinto orden y se verifica que son reconocidos como iguales. Además, se verifica que un triángulo con puntos diferentes no sea reconocido como igual.

Prueba de división de triángulo

Esta prueba verifica la funcionalidad de los métodos que dividen un triángulo en tres y cuatro nuevos triángulos, centrados en un nuevo punto. Se crea un triángulo y se divide, verificando que los tres nuevos triángulos se hayan creado correctamente.

Prueba de carga de una triangulación

Esta prueba se encarga de la validación del proceso de carga de datos y refinamiento de la malla. Se carga una malla inicial, se aplica un proceso de refinamiento con un ángulo mínimo y se verifica que los datos de la malla final sean correctos.

Prueba de Inserción de Puntos Aleatorios

Esta prueba realiza una inserción de puntos aleatorios para evaluar la capacidad del algoritmo para insertar correctamente puntos cuando estos se posicionan muy cerca entre ellos.

5. Arquitectura e implementación del visualizador web en React

5.1. Introducción a React

React es una biblioteca de JavaScript, desarrollada por Meta (ex Facebook), para construir interfaces de usuario en aplicaciones web. Con un enfoque en la creación de componentes reutilizables, React facilita la implementación de aplicaciones de una sola página, ofreciendo un código eficiente y fácil de mantener. Destacan como propiedades su flexibilidad y rendimiento, permitiendo el desarrollo de aplicaciones dinámicas y escalables [13].

5.2. Componentes principales de la aplicación

El visualizador se conforma de tres componentes principales que se relacionan entre si para poder cargar, visualizar y manipular una malla.

- `App.js`: Este es el punto de entrada de la aplicación, administra los estados que representan la triangulación actual, y administra a su vez los componentes `NavHeader` y `Canvas`.
- `NavHeader.js`: Este componente representa el menú de selección de input en formato texto (PSLG). Este componente actualiza la malla actual en `App` para que la pase como parámetro al `Canvas`.
- `Canvas.js`: Este componente usa la librería Three.js para generar un canvas que soporte gráficos en 3D. De esta forma, se puede renderizar la triangulación en el navegador web. Three.js cuenta además con una funcionalidad de controles para manipular la geometría, usando eventos como *click*, *scroll* y arrastrar. De tal manera que se logra una experiencia interactiva permitiendo *zoom*, desplazamiento y rotación de la malla.

En la siguiente imagen se muestra una captura del código de la aplicación:

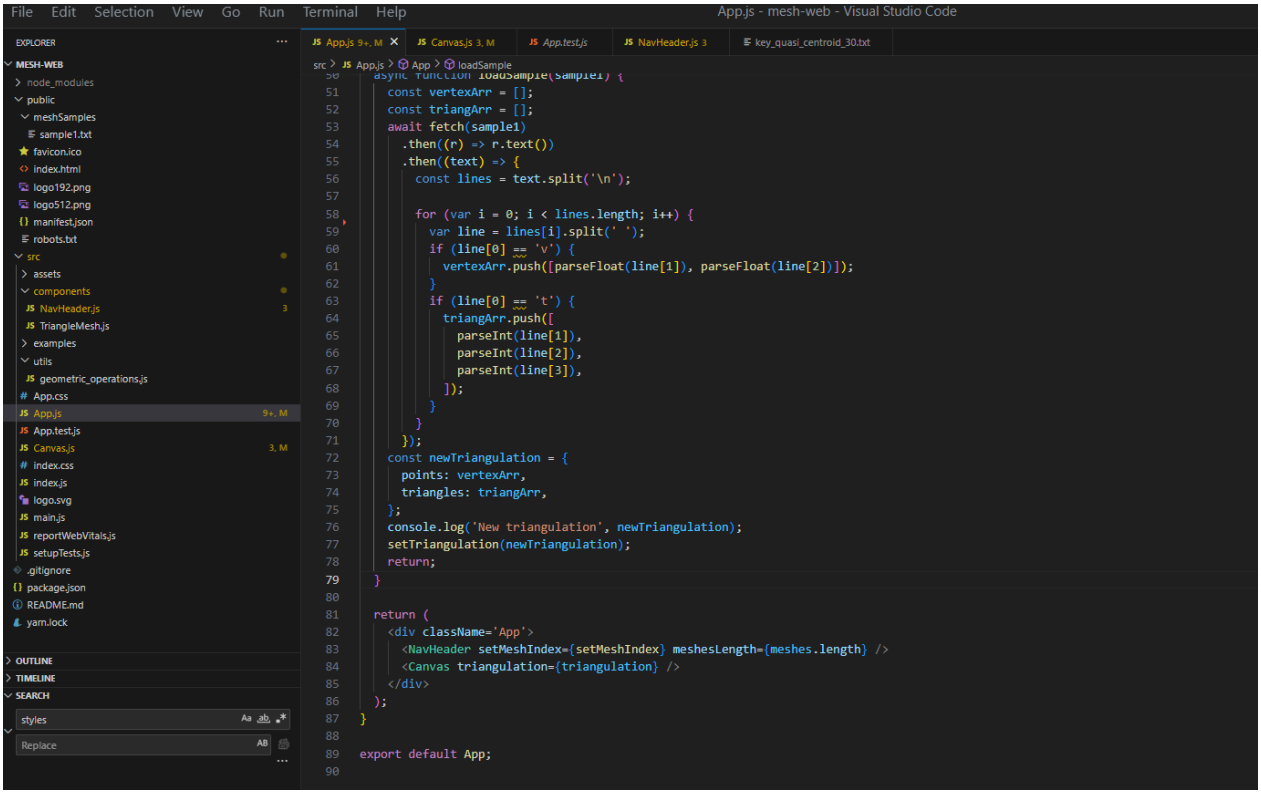


Figura 10: Captura del ambiente de desarrollo de la aplicación. Se trabajó usando el editor de código Visual Studio Code.

Tomando en cuenta los 3 componentes mencionados se cuentan aproximadamente 250 líneas de código escritas.

5.3. Integración de Amplify en el proyecto React

La aplicación de React se alojará en un entorno de preparación en la nube utilizando Amazon Web Services Amplify [1, 2], una plataforma de desarrollo que facilita la implementación de aplicaciones web y móviles. Esta característica permitió proporcionar un entorno de prueba accesible para el uso de la aplicación. Ver Figura 11.

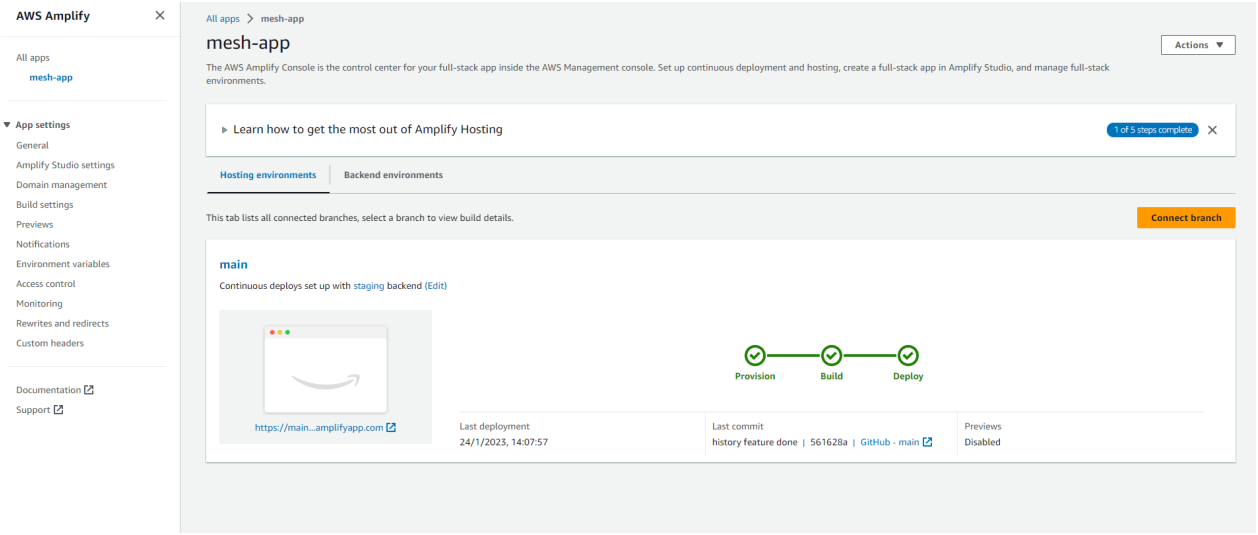


Figura 11: Vista del cliente web de Amplify, donde se aloja la aplicación de React.

6. Resultados y evaluación

En esta sección, se presentan y evalúan los resultados obtenidos a lo largo de este trabajo. Los resultados abarcan el desarrollo e implementación del software en C++ y del visualizador web en React, así como la evaluación de los algoritmos Lepp Delaunay y Lepp Casi Delaunay.

6.1. Software de triangulaciones en C++ y algoritmos Lepp

6.1.1. Geometrías de interés

Se presentan los resultados obtenidos tras la aplicación de los algoritmos Lepp Delaunay y Lepp Casi Delaunay a varias geometrías de interés. Estos resultados proporcionan una comparación cuantitativa de la capacidad de estos algoritmos para refinar las mallas en función del ángulo de refinamiento. Las Tablas 1 y 2 representan el tamaño final de las mallas, en términos de la cantidad de triángulos, para cada uno de los algoritmos y diferentes valores de Θ_{tol}.

Posteriormente, se proporcionan visualizaciones de las geometrías para una mayor claridad y para proporcionar una interpretación visual de los datos tabulados.

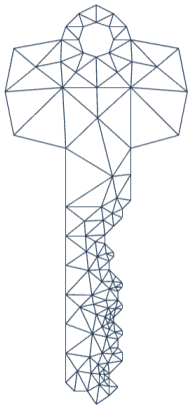
Tabla 1: Algoritmo Lepp Delaunay con inserción del centroide

Geometría	**Ángulo de refinamiento Θ_{tol} en grados**						
	Inicial	**30**	**31**	**32**	**33**	**34**	**35**
Superior	528	1854	1951	2120	2365	2784	3197
Key	54	177	182	199	232	279	312
Neuss	3070	8491	8905	9476	10211	11292	12908
Lewis	4300	25768	27744	30001	34176	39516	48101

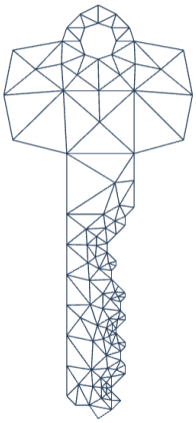
Tabla 2: Algoritmo Lepp Casi Delaunay con inserción del centroide

Geometria	**Ángulo de refinamiento Θ_{tol} en grados**						
	Inicial	**30**	**31**	**32**	**33**	**34**	**35**
Superior	528	1828	1914	2078	2271	2678	3193
Key	54	175	188	200	226	257	392
Neuss	3070	8437	8890	9540	10223	11365	13504
Lewis	4300	26297	28083	30497	34271	39764	49801

Key



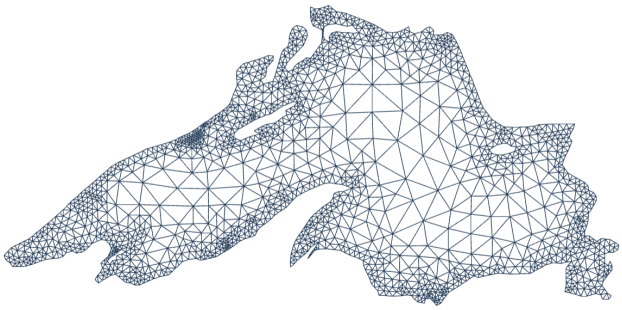
(a) Geometría refinada con algoritmo **Lepp Delaunay** con θ igual a 30 grados.



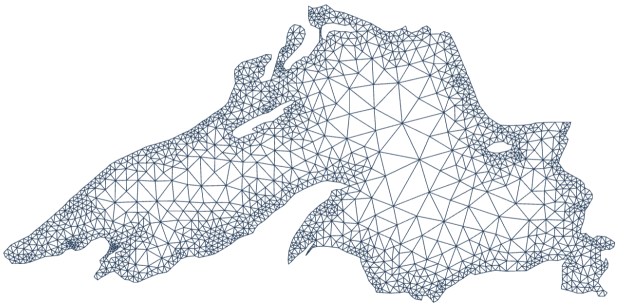
(b) Geometría refinada con algoritmo **Lepp Casi Delaunay** con θ igual a 30 grados.

Figura 12: Comparación de geometrías refinadas con diferentes algoritmos con θ igual a 30 grados.

Superior Lake



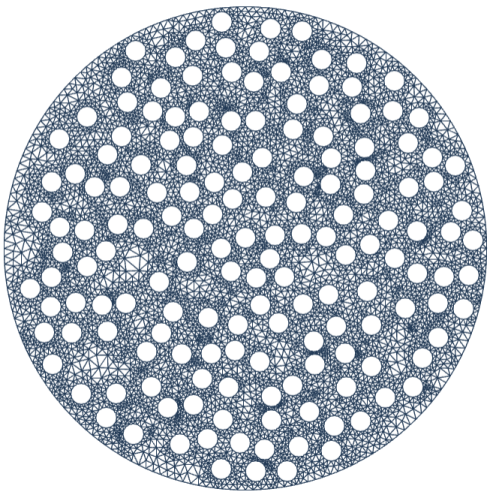
(a) Geometría refinada con algoritmo **Lepp Delaunay** con θ igual a 35 grados.



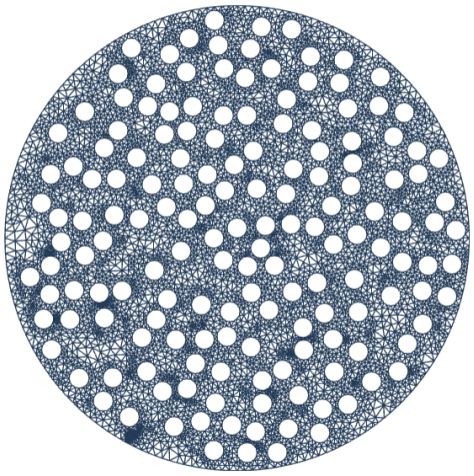
(b) Geometría refinada con algoritmo **Lepp Casi Delaunay** con θ igual a 35 grados.

Figura 13: Comparación de geometrías refinadas del Superior Lake con diferentes algoritmos con θ igual a 35 grados.

Neuss



(a) Geometría refinada con algoritmo **Lepp Delaunay** con θ igual a 35 grados.



(b) Geometría refinada con algoritmo **Lepp Casi Delaunay** con θ igual a 35 grados.

Figura 14: Comparación de geometrías refinadas de Neuss con diferentes algoritmos con θ igual a 35 grados.

Lewis

(a) Geometría refinada con algoritmo **Lepp Delaunay** con θ igual a 30 grados.

(b) Geometría refinada con algoritmo **Lepp Casi Delaunay** con θ igual a 30 grados.

Figura 15: Comparación de geometrías refinadas de Lewis con diferentes algoritmos con θ igual a 30 grados.

6.1.2. Evaluación de los algoritmos de triangulación de Delaunay y Casi Delaunay

Calidad de la geometría

A través de la aplicación de los algoritmos en las distintas geometrías, se pudo comprobar la eficacia del algoritmo Casi Delaunay en términos de convergencia. Este algoritmo mostró ser capaz de mejorar todos los triángulos de la malla por encima de la tolerancia θ, independientemente del conjunto de datos en el que se aplicaba. Este hallazgo valida la robustez del algoritmo Casi Delaunay, y su aplicabilidad para el refinamiento de mallas en una variedad de geometrías.

A continuación, se presenta una serie de gráficos de barras que comparan el rendimiento de los algoritmos de Delaunay y Casi Delaunay para cada una de las geometrías estudiadas. Cada gráfico muestra el número total de triángulos en la malla final obtenida con cada algoritmo para diversos ángulos de refinamiento. Los valores representados en las barras corresponden al número total de triángulos generados por los respectivos algoritmos de triangulación.

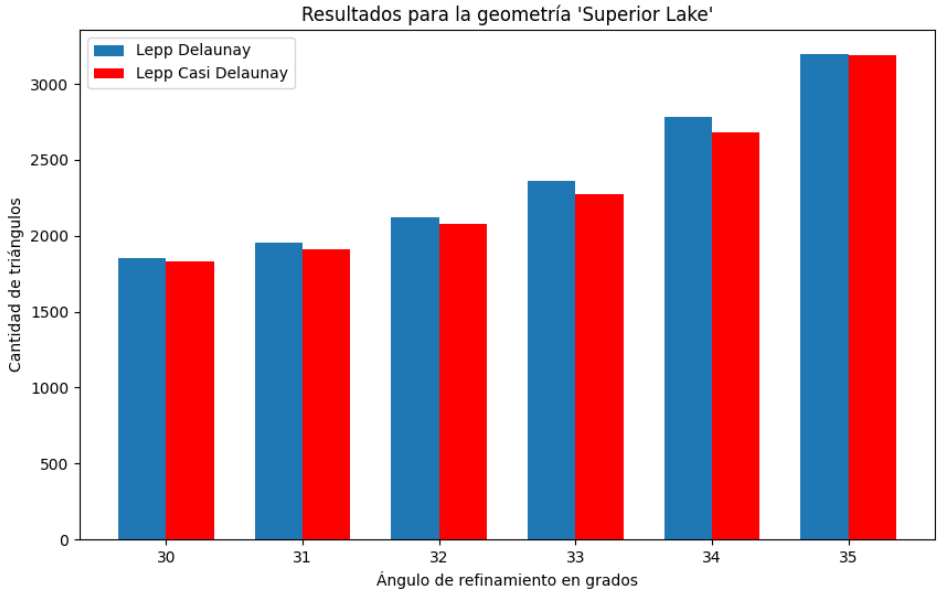


Figura 16: Comparación de los algoritmos para la geometría Superior Lake.

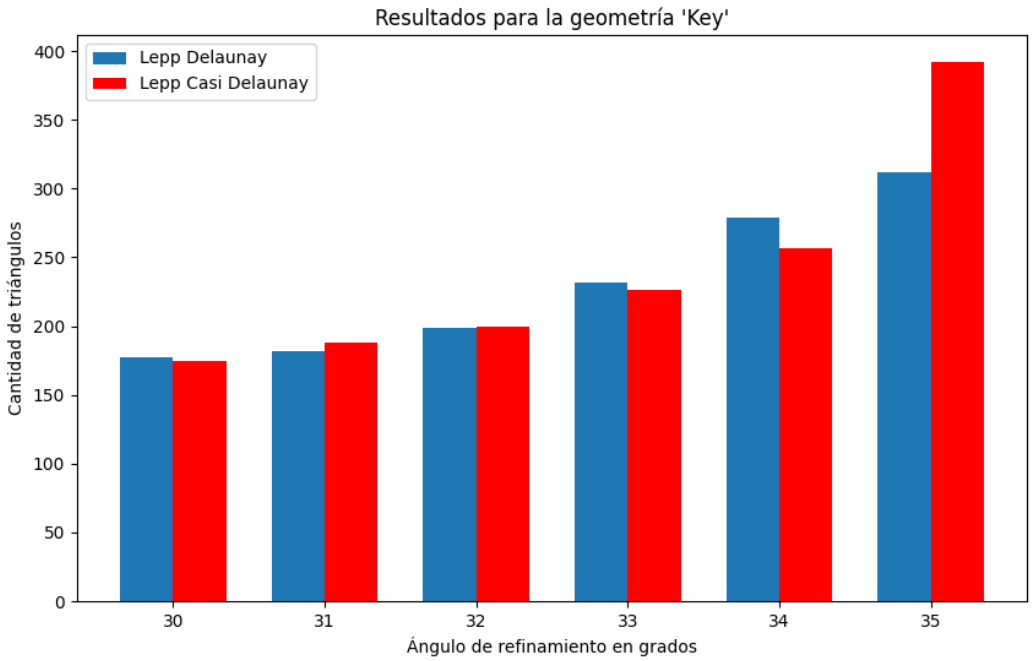


Figura 17: Comparación de los algoritmos para la geometría Key.

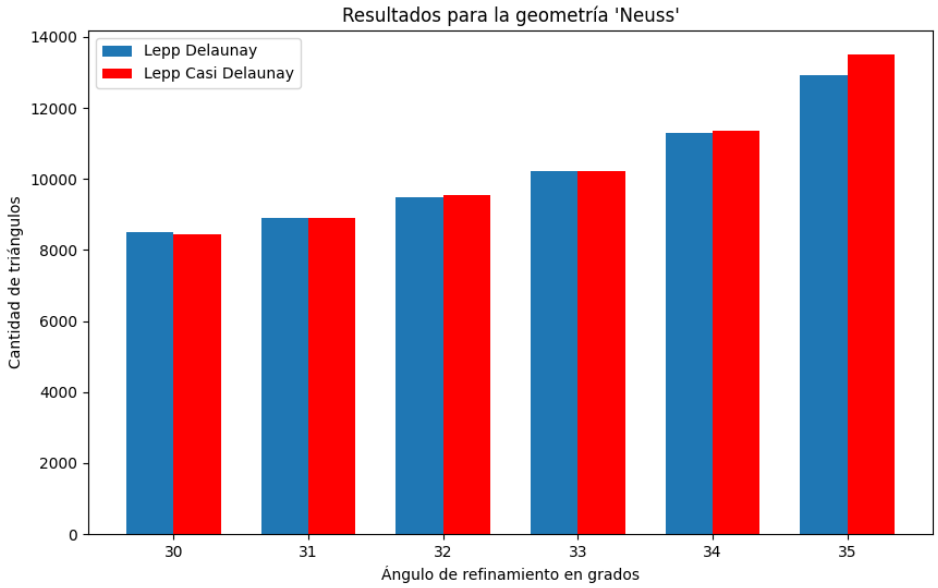


Figura 18: Comparación de los algoritmos para la geometría Neuss.

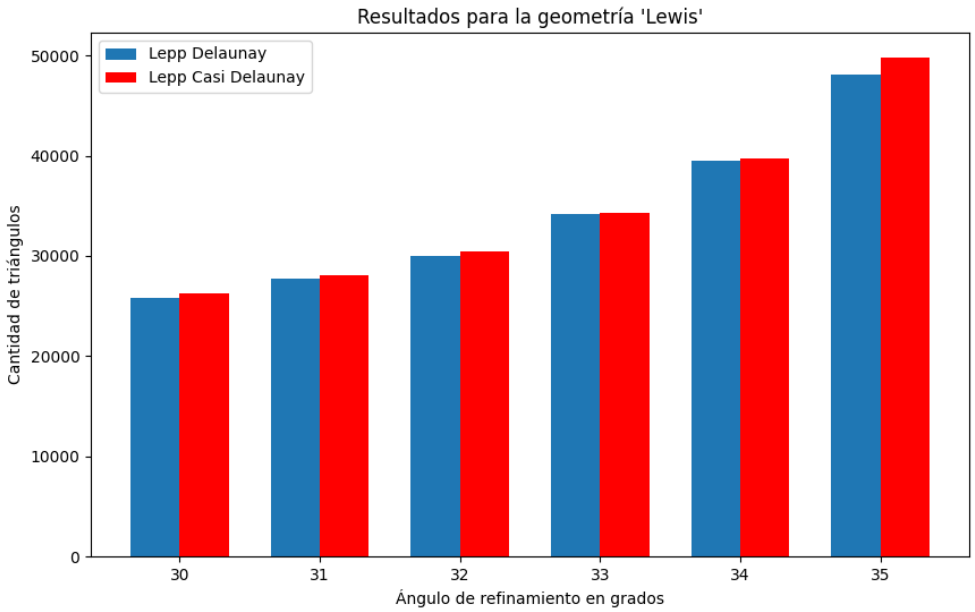


Figura 19: Comparación de los algoritmos para la geometría Lewis.

En términos geométricos y cuantitativos, observamos que la malla refinada mediante el algoritmo Lepp Casi Delaunay se comportó de manera similar a la malla refinada con el algoritmo Lepp Delaunay. Esto no sólo se evidencia en la forma y disposición de los triángulos en las mallas, sino también en la cantidad de triángulos generados durante el proceso de refinamiento.

Desempeño de implementación de software en C++ versus Python

Para evaluar y contrastar el desempeño entre las implementaciones en C++ y Python del proceso de refinamiento de mallas, se diseñó un experimento específico. Este fue ejecutado en una notebook HP equipada con un procesador AMD Ryzen 7 4800H y 8GB RAM. La base de la comparativa se centró en medir los tiempos de ejecución que las implementaciones tardan en refinar una triangulación Delaunay inicial bajo las mismas condiciones de hardware. La triangulación usada para este experimento es *Superior Lake*, donde se promedió el tiempo de refinamiento de 10 ejecuciones para cada ángulo de tolerancia y algoritmo de refinamiento usando la inserción del centroide.

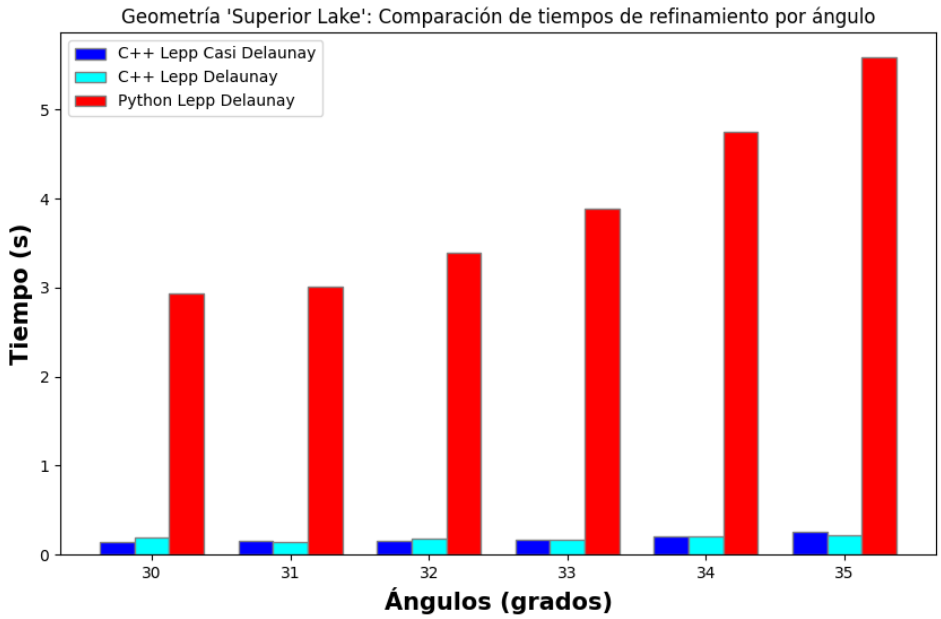


Figura 20: Comparación de tiempos de ejecución promedio para geometría Superior Lake en segundos. Los ángulos de tolerancia de refinamiento van desde 30 hasta 35 grados.

A partir de estos resultados, se observa que la implementación en C++ ofrece una ventaja significativa en términos de tiempo de ejecución en comparación con la versión en Python. Esto resalta la importancia de elegir el lenguaje e implementación adecuados según los objetivos y necesidades del estudio en particular.

Por ejemplo, si se deseara ejecutar este software desde un servidor, el uso de recursos se vuelve un factor a considerar. En muchos servicios de hosting o plataformas en la nube, los costos se asocian directamente con el tiempo de procesamiento y uso de recursos. Un software que demore más en realizar una tarea podría traducirse en gastos significativamente mayores, especialmente cuando se trata de operaciones que se ejecutan repetidamente o por largos períodos de tiempo.

Además, es relevante mencionar que en el mismo computador HP con la implementación de C++ se pudo refinar una malla inicial del orden de 100,000 puntos con valores aleatorios dada una tolerancia de 35 grados, mientras que la versión de Python pudo refinar una malla inicial de hasta 10,000 puntos con valores aleatorios.

Finalmente, el uso de aritmética adaptativa permitió que para las geometrías usadas siempre la malla refinada fuera del mismo tamaño, en cantidad de puntos y triángulos (por propiedad del algoritmo), dado un ángulo de tolerancia y algoritmo de refinamiento específico. Situación que no ocurre en la versión de Python en los que el tamaño de la malla final refinada varía en cada ejecución.

6.1.3. Limitaciones

Una limitación de este software es que, en su estado actual, no maneja aristas restringidas interiores ni ángulos restringidos interiores formados a partir de dos aristas restringidas. Esto significa que no puede gestionar directamente la inserción de aristas restringidas de un PSLG. En lugar de ello, el software maneja únicamente los bordes de la triangulación, que representan los contornos externos o internos de la triangulación.

Sin embargo, cabe mencionar que esta limitación no impide refinar PSLG ya definidos. Esto significa que, aunque el software no puede insertar ni definir aristas restringidas internas, sí puede mejorar la calidad de la malla dentro de un PSLG con bordes definidos.

6.2. Visualizador Web desarrollado en React

Como parte de este trabajo, se implementó el visualizador web para la visualización e interacción con triangulaciones.

En el visualizador, los usuarios pueden cargar archivos de malla generados localmente para su visualización y análisis, explorando detalles visuales de la malla y manipulándola mediante transformaciones como *zoom* o desplazamiento.

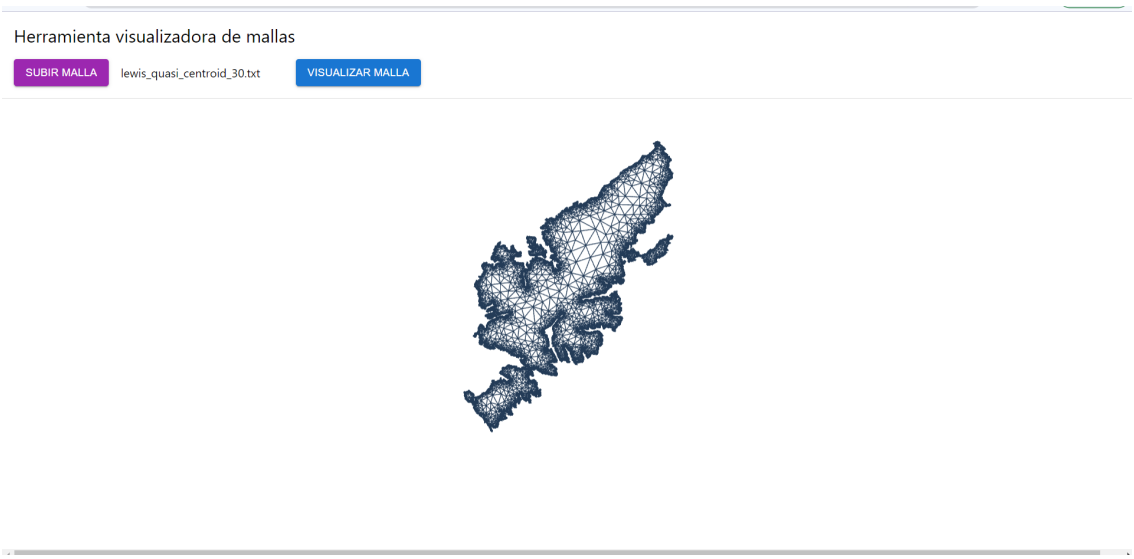


Figura 21: Interfaz de usuario del visualizador con una geometría Lewis cargada.

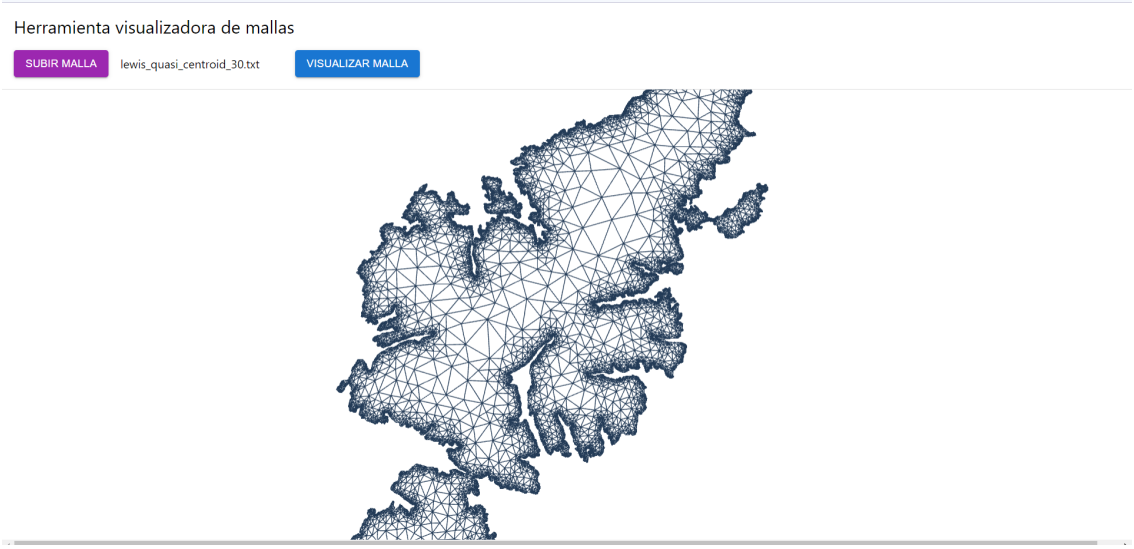


Figura 22: Geometría Lewis con zoom en el visualizador web.

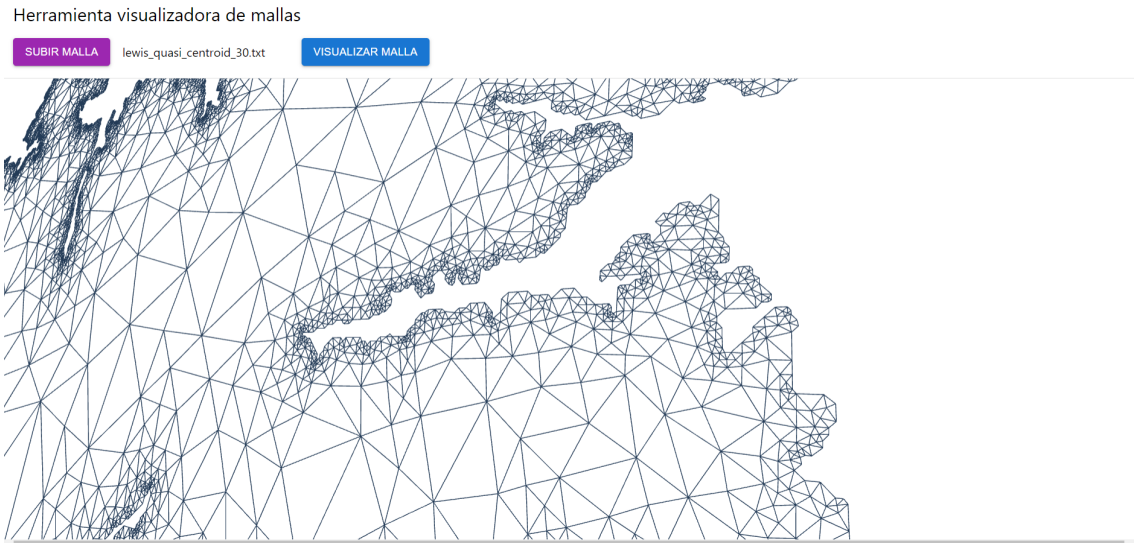


Figura 23: Geometría Lewis con zoom, rotación y traslación en el visualizador web.

6.2.1. Limitaciones

Una de las principales limitaciones radica en el hecho de que el sistema de software matemático no está alojado en la nube. Esto significa que la generación de mallas debe realizarse localmente en el equipo del usuario. Una vez generada la malla, el usuario puede cargarla en la aplicación para su visualización y análisis.

7. Conclusiones y trabajo futuro

7.1. Conclusiones

Este trabajo abordó la investigación y comparación de los algoritmos Lepp Delaunay y Casi Delaunay, en la generación y refinamiento de mallas para distintos conjuntos de datos. Los algoritmos fueron implementados y probados usando el software en C++, y se desarrolló un visualizador web para facilitar la presentación y comparación de los resultados.

En primer lugar, la implementación del software demostró la viabilidad de aplicar estos algoritmos para el manejo de distintas geometrías. Además, la decisión de implementar el software en C++ permitió aprovechar el manejo de memoria del lenguaje y bibliotecas de aritmética adaptativa. De esta forma, se logró construir una herramienta que permite:

1. Generar triangulaciones Delaunay a partir de un PSLG.
2. Hacer cálculo de operaciones mediante aritmética adaptativa.
3. Refinar una triangulación usando el algoritmo Lepp Delaunay con una tolerancia de hasta 35°.
4. Refinar una triangulación usando el algoritmo Lepp Casi Delaunay con una tolerancia de hasta 35°.

Adicionalmente, el visualizador web proporcionó una herramienta para analizar y comparar las mallas generadas. Su capacidad para mostrar las diferencias visuales entre las mallas Delaunay y Casi Delaunay ayudó a obtener una comprensión de las similitudes y diferencias de estos algoritmos.

Finalmente, se comprobó que el algoritmo Casi Delaunay es capaz de converger y mejorar todos los triángulos de la malla por encima de una tolerancia θ específica de hasta 35°. Las mallas generadas por el algoritmo Casi Delaunay resultaron ser comparables a las generadas por el algoritmo Delaunay tanto en términos de propiedades geométricas como en cantidad de triángulos.

7.2. Trabajo futuro

Este trabajo da la oportunidad para ser extendido y que puedan abordarse algunas de las limitaciones de la implementación actual del software, para así desarrollar características adicionales que amplíen su aplicabilidad y eficiencia.

Una de las posibles mejoras y funcionalidades a implementar en el software

es el de manejo de aristas restringidas.

Además, otra área a mejorar se encuentra en el ámbito del sistema web, ya que actualmente la generación de mallas se realiza localmente usando el software de C++, y los resultados obtenidos se cargan manualmente posteriormente en la aplicación web, limitando el uso del software. Para un futuro, se propone implementar el software matemático en un ambiente en la nube. De esta manera, la generación de mallas podría realizarse directamente en la nube, lo que permitiría un sistema web completamente funcional y accesible desde cualquier lugar, proporcionando un enfoque más integrado en la generación y visualización de mallas.

Un aspecto final que podría ser investigado y desarrollado en un futuro es la exploración de algoritmos paralelos para el refinamiento de mallas, especialmente aquellos diseñados para ejecución en GPU. Las capacidades de cómputo paralelo de las tarjetas gráficas modernas ofrecen la posibilidad de manejar y procesar grandes mallas con eficiencia.

Estos avances podrían llevar a la generación y refinamiento de mallas a un nivel más profundo, proporcionando una herramienta flexible que pueda ser aplicada en un rango más amplio de casos y dominios.

Bibliografía

[1] Amplify: *Servicio de construcción y alojamiento de aplicaciones web.* https://aws.amazon.com/es/amplify/.

[2] AWS: *Servicio de cloud computing.* https://aws.amazon.com/es/.

[3] Beazley, David y Brian K. Jones: *Python: CookBook, Third Edition.* O' Reilly, 2013.

[4] Bedregal, C. y M.c. Rivara.: *New results on Lepp-Delaunay algorithms for quality triangulations.* Procedia Engineering, 2014.

[5] Berg, Mark de y et. al: *Computational Geometry: Algorithms and Applications, Third Edition. Capítulos 1, 3 y 9,.* Springer-Verlag Berlin Heidelberg, 1997/2008.

[6] Bern y Eppstein: *Mesh Generation And Optimal Triangulation.* Lecture Notes Series on Computing in Euclidean Geometry, pp. 23-90, 1992.

[7] CGAL: *The Computational Geometry Algorithms Library.* https://www.cgal.org/.

[8] Chew, L. Paul: *Guaranteed-quality Triangular Meshes.* Department of Computer Cience, Cornell University, 1989.

[9] CMAKE: *CMake: Herramienta empaquetadora de software.* https://cmake.org/.

[10] Díaz, Javier: *Herramienta de resolución de triangulaciones geométricas.* Memoria para optar al título de ingeniero, 2018.

[11] Google: *Google Test: Framework de pruebas unitarias desarrollado por Google para C++.* https://github.com/google/googletest.

[12] JetBrains: *CLion: IDE multiplataforma para C y C++.* https://www.jetbrains.com/es-es/clion/.

[13] React: *Framework de desarrollo web basado en Javascript.* https://es.reactjs.org/.

[14] Rivara y Calderón: *Lepp Terminal Centroid Method for Quality Triangu-*

lation: A Study on a New Algorithm. Department of Computer Science, University of Chile, 2008.

[15] Rivara, MC: *New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations, International Journal for Numerical Methods in Engineering 3313-3324.* 1997.

[16] Rivara, MC y Javier Díaz: *Terminal Triangles Centroid Algorithms for Quality Delaunay Triangulation, Des. 125: 102870.* Comput. Aided, 2020.

[17] Rivara, M.C. y N. Hitschfeld: *Lepp-delaunay algorithm: a robust tool for producing sizeoptimal quality triangulations.* 2000.

[18] Rivara, MC y Mellado: *Quasi Delaunay Lepp algorithms for quality triangulation.* En preparación, 2023.

[19] Rodriguez, Pedro: *Parallel lepp-based algorithms for the generation and refinement of triangulations.* Tesis de Doctorado, Universidad de Chile, 2015.

[20] Ruppert, Jim: *A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation.* Journal of Algorithms, Volume 18, Issue 3, May 1995, 1995.

[21] Schirra, S.: *Precision and robustness in geometric computations.* Algorithmic Foundations of Geographic Information Systems., 1997.

[22] Shewchuck, J.: *Mesh generation for domains with small angles.* In Proc. 16th Annu. ACM Sympos. Comput. Geom., pages 1–10, 2000.

[23] Shewchuck, J. R.: *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates.* 1997.

[24] Shewchuk, Jonathan: *Triangle: A two-dimensional quality mesh generator and delaunay triangulator.* https://www.cs.cmu.edu/ quake/triangle.html.

[25] Stroustrup, Bjarne: *The C++ Programming Language, 4th Edition.* Addision-Wesley, 2013.

[26] Three.js: *Three.js: Librería gráfica orientada a la web.* https://threejs.org/.